Parveen Khurana

124 Followers About Following

Regularization Methods

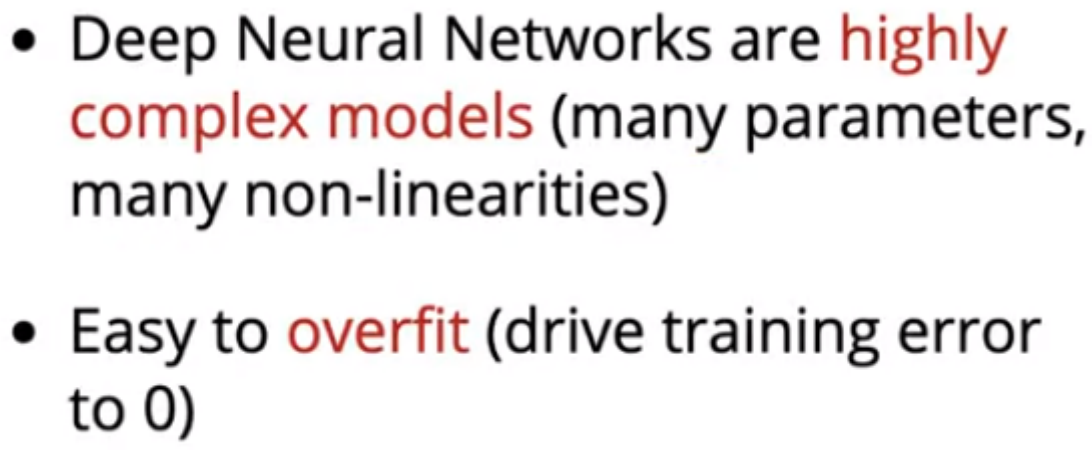
P Parveen Khurana Feb 7, 2020 · 10 min read

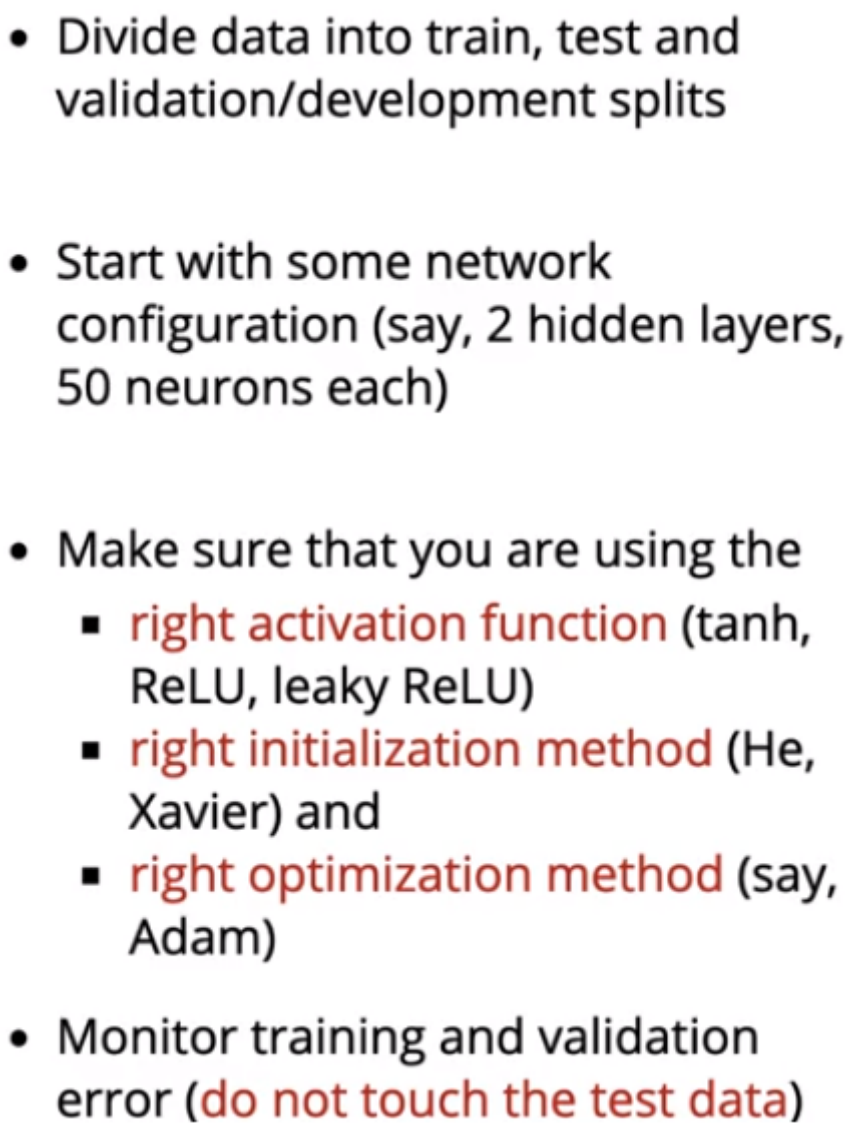
This article covers the content discussed in the Regularization Methods module of the Deep Learning course and all the images are taken from the same module.

Overfitting in DNNs:

Deep Neural Networks are highly complex models and will suffer from the problem of high variance(as discussed in this article) and will try their best to drive the training error to 0 by completely fitting the training data but when we try this model on the test data or the validation data, it will give a very high error.

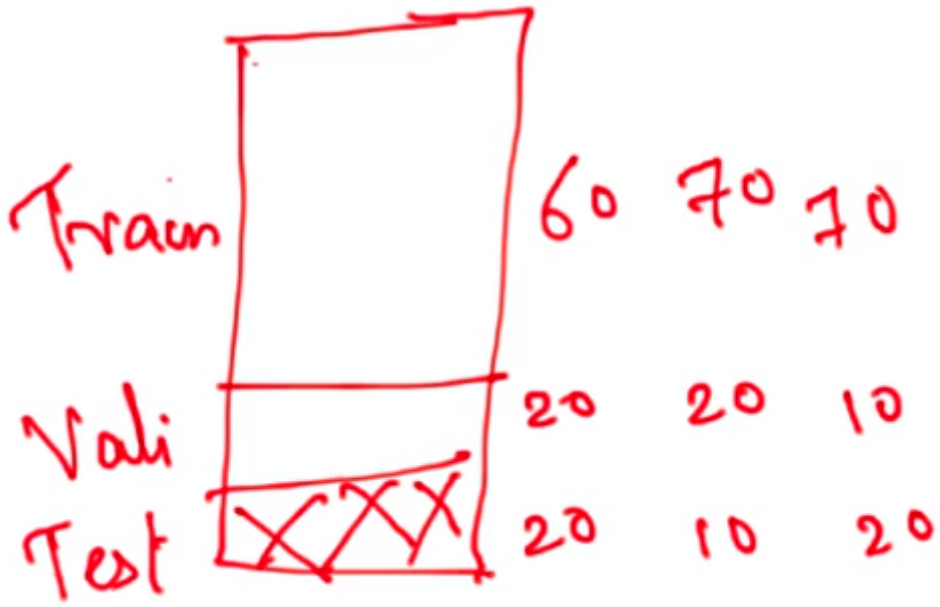
So, we need to deal with the complexity of the model and make sure that the complexity of the model is somewhere around the sweet spot which is the ideal model complexity where the bias, as well as the variance both, are low and which gives us low test error.





Whatever the **data** is given to us, we should **divide it into three parts**: **training, test, and validation**. The typical ratios of the division are 60, 20, 20 or 70, 20, 10 or 70, 10, 20.

- **Test data** is strictly out of bounds, we should never see it, only at the end when we have done everything, we should test the model with the test data.
- **Training data** is something that we use to train the model to optimize/minimize the training error or the loss function using which we try to learn the parameters of the model.
- **Validation data** is something that we should keep track of to make sure that the model is not too complex where even you have over-minimized the training loss, it



Let's see how we use the training and validation splits:

We start with some training configuration and we need to ensure the following:

- That the right activation functions are used(we can use **ReLU**, **Leaky ReLU** with Feed Forward Neural Network, Convolutional Neural Network and we can use **tanh** with the Recurrent Neural Networks and Feed Forward Neural Network)
- That we are using the right initialization method for the parameters and not initializing them to all 0 or very large or all equal
- That we are using right optimization method and here **Adam** is the default choice.

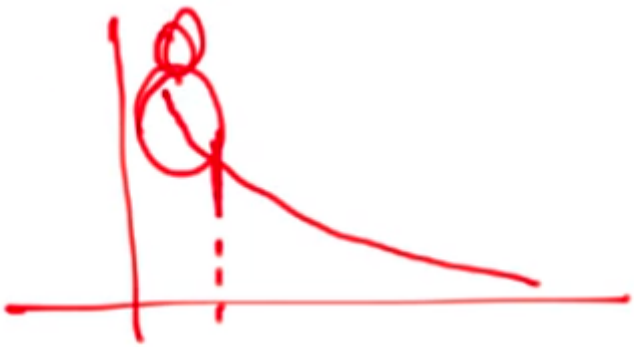
Now with these settings, we start training the model and monitor the training error and the validation error. We can plot these two **errors on the y-axis** with the **number of epochs on the x-axis**. Now there are different scenarios here:

- both the training error and the validation error could be high
- training error could be low but validation error could be high
- both the training error and the validation error could be low

So, we are typically left with three situations: both the errors are high, both are low or the training error is low and the validation error is high. Let's see what we should do in each of these cases:

Training Error	Valid Error	Cause	Solution
High	High	High bias	- Increase model complexity - Train for more epochs
Low	High	High variance	- Add more training data (e.g., **dataset augmentation**) - Use **regularization** - User **early stopping** (train less)
Low	Low	Perfect tradeoff	- You are done!

Case 1: If the training error itself is high, that means we are lying in the region of high bias, model is not complex enough so we could try to change the number of hidden layers, we could change the number of neurons in hidden layers or it might be the case that we have chosen a reasonably complex model but have not trained it enough

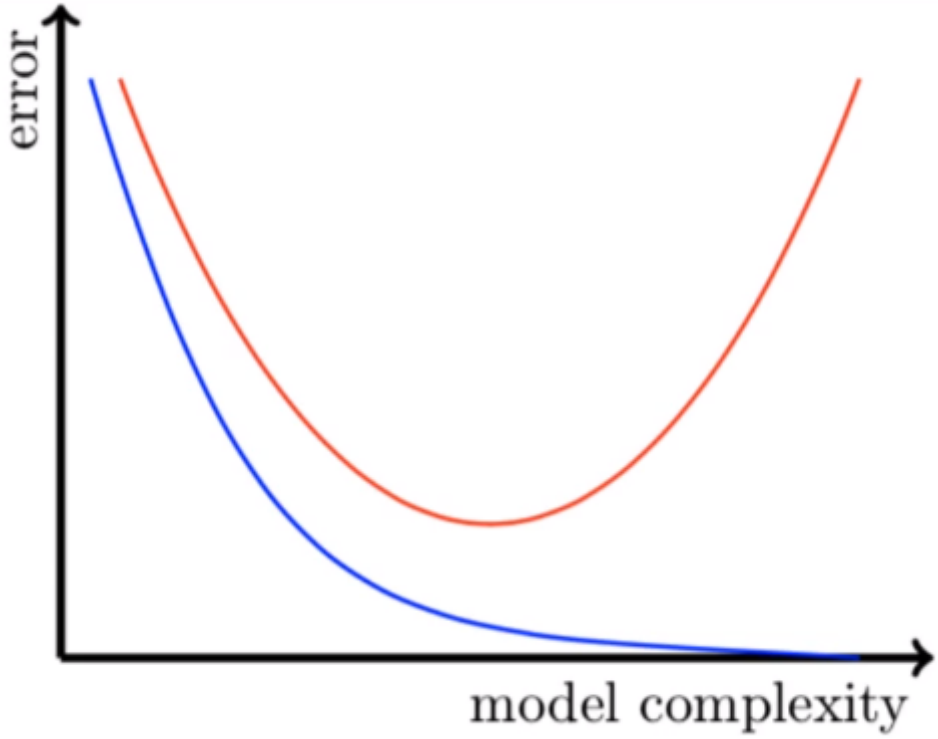


Case 2: The training error is low but the validation error is still high then we have the issue of high variance. We could use any of the following 3 popular options: we could go with dataset augmentation, we could use regularization or we could use early stopping. We will see these three methods below in this article.

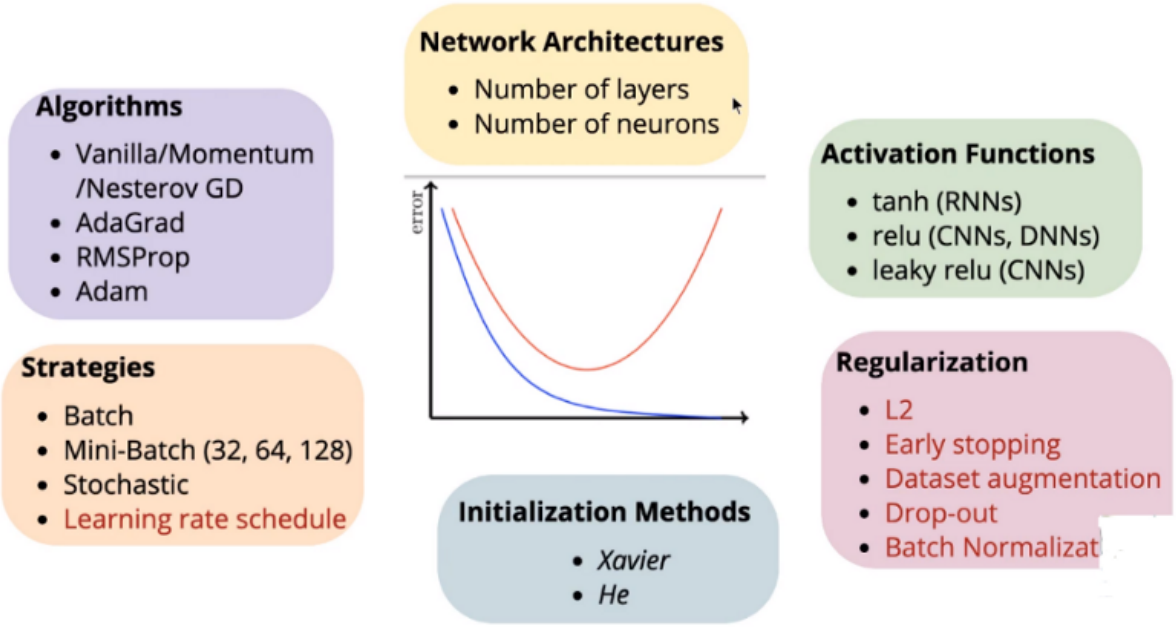
A detour into hyperparameter tuning:

In Deep Learning, we have several choices to make. One of those choices, when we are using Feed Forward Neural Network or CNN, is to decide how many layers(we typically mean the hidden layers as the input and the output layer would always be there) to use, how many neurons to use in each layer. Then, the other set of choices that we need to make is what Algorithms to use, what strategy to use like Batch, mini-batch, stochastic, what learning rate to use, then we have the different choices for the Activation functions, we have options for parameters initialization methods, then we have the several regularization techniques which we discuss below in this article, we can use these regularization techniques together or separately so the choice to make is which of these combinations of hyper-parameters to try on.

The way we make all these choices is that we monitor the validation error(blue curve in the below image) and the training error(red curve in the below image). For example whether to use Adagrad or RMSProp or some other algorithm, we see how the training error is behaving with these algorithms, if we are using Vanilla GD maybe its stuck in some local flat regions and that might be the case why training error is not moving so instead of training for more epochs we can try out Adam which is less sensitive to this flat region problem.

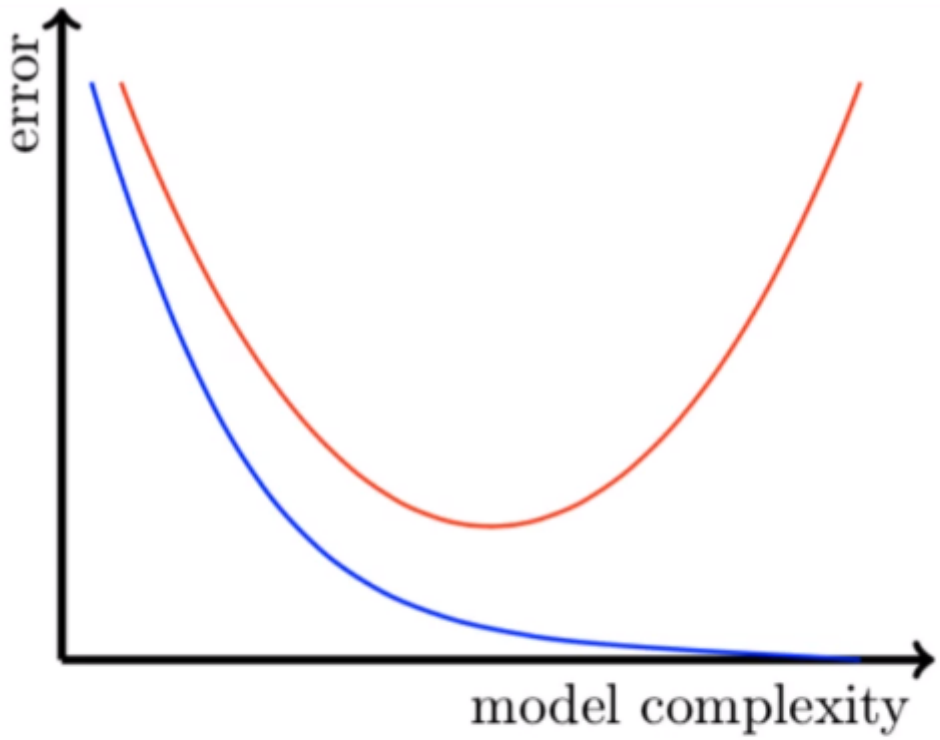


to figure out) like what is the right configuration of the layers and the neurons, which is the best Learning Algorithm for the given task and so on. We are not going to learn these Hyper-parameters using data but instead, we are going to run some experiments, note down these errors and then decide which one is the best.



L2 Regularization:

The issue that we have is that the training error would be very low with DNNs but the validation error is staying high.



We want to achieve the sweet spot in terms of model complexity where the training error is not completely zero but the validation is also reasonably small. Let's see how to do that:

The training error would be given as:

$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N}(y_i - \hat{f}(x_i))^2$$

We try to minimize the loss with respect to **θ** where **θ represents all the parameters** of the network:

$$\theta = [W_{111}, W_{112} + \cdots + W_{Lnk}]$$

Now instead of using the above one equation the loss function, we use the below loss function which is the original loss function plus some quantity say **P**:

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

but **P** is high because if that happens then the total quantity which is the new loss is going to be high. So, we are adding some constraints which will prevent the model from driving the training error to 0.

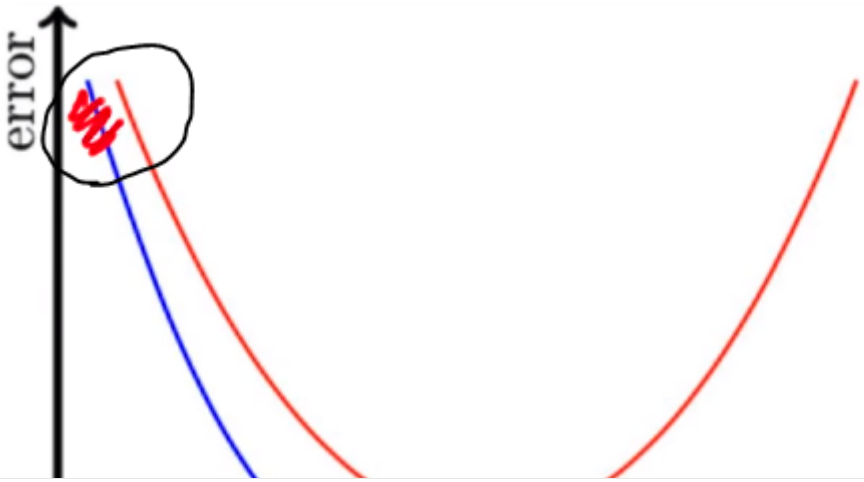
This quantity **P** is just the squared sum of all the parameters:

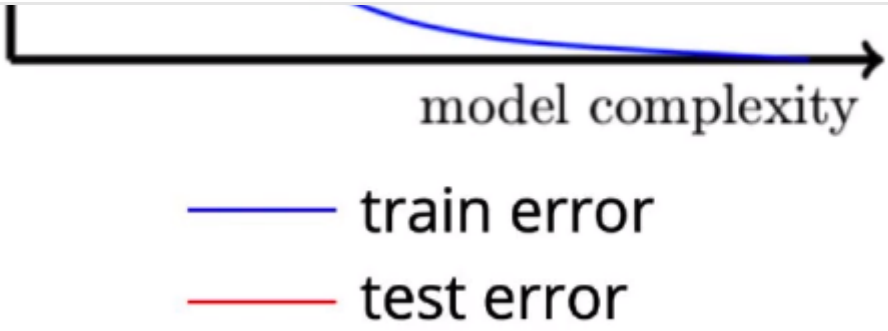
$$\Omega(\theta) = ||\theta||_2^2$$
$$= W_{111}^2 + W_{112}^2 + \cdots$$

If we set all the weights to 0, in that case, **P** would be 0, but if all the weights are 0 that means the model has not learned anything, it has not adjusted the parameters so the yellow highlighted quantity in the below image is going to be high and hence the overall loss is going to be high

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

And with the overall loss being high, we would lie somewhere in the high bias region(below image)



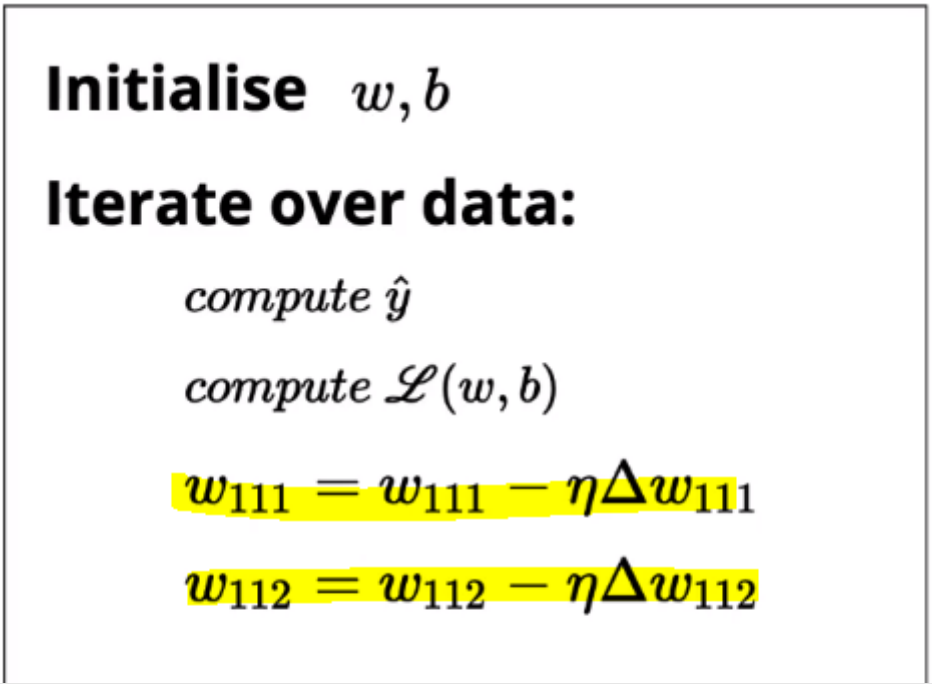


Now let's see the other case where we try to make the original loss to a minimum, in that case, what might happen is that some of the weights may take on a large value because of which the quantity **P** would be high which implies the overall new loss would be high and the same story repeats.

So, now our objective is to drive the training error to zero but to also ensure that the weights do not grow too large so we are placing a constraint on them. Under this constraint, the training error would not go up to 0, it will land somewhere in the sweet spot and beyond that, if we try to reduce the loss then the quantity **P** will grow up and the overall loss would be high.

Let' see how to use GD with this new loss into consideration:

We update any of the weights with respect to the derivative of the loss function with respect to that weight.



$w_{313} = w_{313} - \eta\Delta w_{313}$

till satisfied

Now the derivative of the loss function with respect to any weight is given as:

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$\Omega(\theta) = W_{111}^2 + W_{112}^2 + \cdots + W_{Lnk}^2$$

$$\Delta W_{ijk} = \frac{\partial \mathscr{L}(\theta)}{\partial W_{ijk}}$$

$$= \frac{\partial \mathscr{L}_{train}(\theta)}{\partial W_{ijk}} + \frac{\partial \Omega(\theta)}{\partial W_{ijk}}$$

We already know how to compute the quantity in red in the above image. The other quantity in blue would just be:

‘2Wijk’

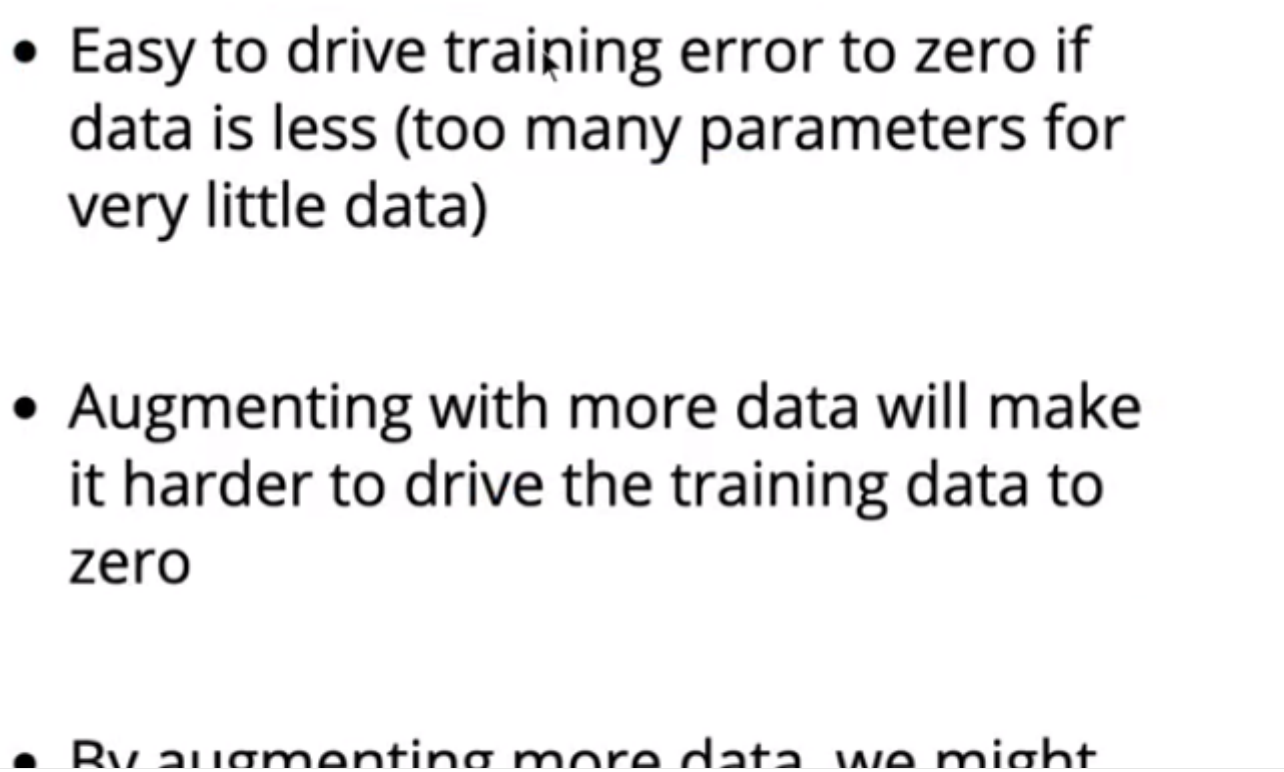
$\nabla L_{train}(\theta)$

Dataset Augmentation:

The problem that we have at hand is that a DNN would drive the training error to 0. If we have very small training data set, then we would see overfitting as DNN has many many parameters and overall training loss would be close to 0 but the validation loss would be high as the model is trained on a small dataset on which the model completely overfit the data and the model has not seen data points outside the training data set.

So, to deal with this we try to Augment more data. In many cases, data may not be available as we need not only **'x'** but we need the **(x, y)** pairs so we not only need the image, we need the label associated with it as well whether it's a mountain image, beach image or so on and someone has to label this and labeling often is expensive. So, what we do in practice is to use **dataset augmentation**.

For the given image, we could blur it a bit, we could crop it or we could take half portion of the image and things like that like we could rotate the image a bit, we could translate it a bit, and we get a new image and the **'y'** label remains the same as the original image has been used to get this transformed image and we could use the same **'y'** label for that.



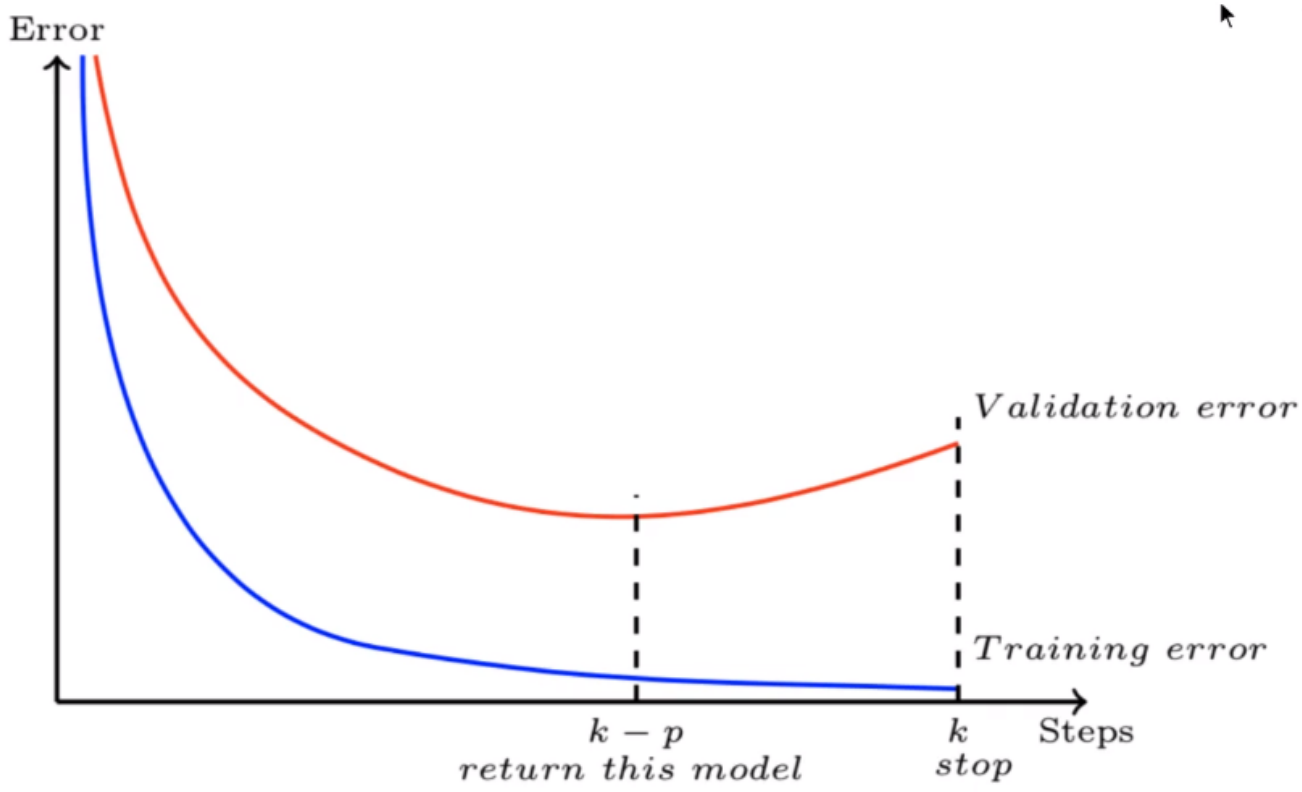
to valid/test data (hence, effectively reduce the valid/test data)

Early Stopping:

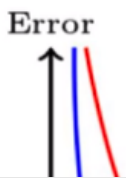
We keep training our model for a large number of epochs and we keep monitoring the loss. We have a **patience parameter** denoted by **'p'**

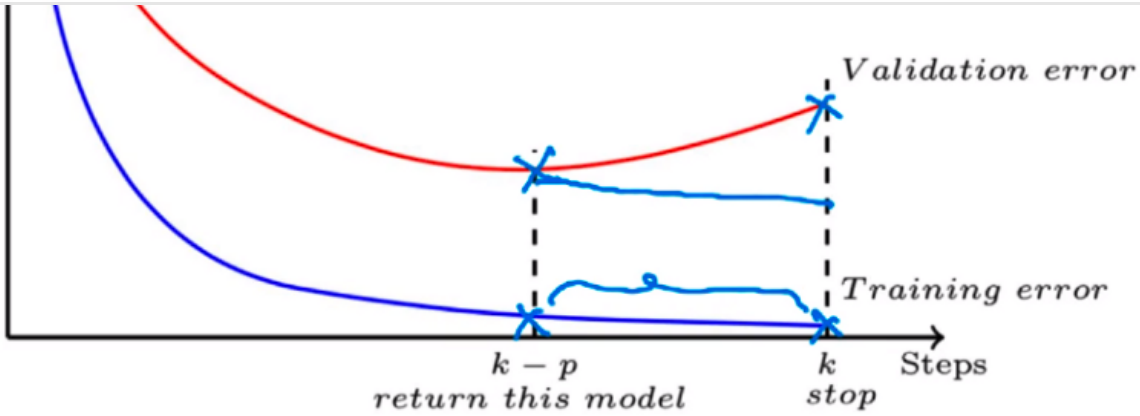
say **p = 5 epochs**

We will plot out the training and the validation error after every epoch.



Now we will wait for some epochs(as per patience parameter) and see if the validation error decreases any further which is clearly not happening in this case as is clear in the below image.





Whatever validation we had achieved at **'k-p'** epochs, it either increased after **'p'** epochs or stayed the same, either of this is possible. Now as we are training the model more and more, the training error is decreasing but the validation error is increasing(in the above case).

We train for a large number of epochs, set some value for the patience parameter depending on the amount of data we have, and if for the **'patience parameter number of epochs'** the validation error is not decreasing even though the training error is decreasing(exactly the situation as in the above case), we say we have lost patience, there is no use of training the model anymore as the model is just memorizing the data given to it and doing no better on the validation set. And we stop the training there itself. So, wherever the last minimum error we got, we are going to return that model parameter(we will save the model parameters after each epoch).

And whichever of these epochs gives us the minimum validation loss, that is the epoch for which we will return the model's weights.

In summary, we know that DNNs are very complex models and they drive the training error to 0 very readily and to avoid that we need to have everything in order from the start i.e we must right set of hyper-parameters and we must use the appropriate Regularization technique as per the task at hand.

Deep Learning Regularization Hyperparameter Tuning Artificial Neural Network

Artificial Intelligence